

hereby certify that this correspondence is being deposited in the United States Postal Service as  
 express mail in an envelope addressed to: Commissioner of Patents and Trademarks, Washington,  
 C., 20231, on June 11, 1996. Express Mail Receipt No. 6 30 43 8 US

June 11, 1996  
 Date of Signature

Robert Tanker

PIPELINED INSTRUCTION DISPATCH UNIT  
 IN A SUPERSCALAR PROCESSOR

RECEIVED

AUG 4 2000

OFFICE OF PETITIONS

Marc Tremblay

IS  
B

5

BACKGROUND OF THE INVENTION

1. Field of the Invention

This invention relates to computer architecture.  
 In particular, this invention relates to the design of  
 an instruction unit in a superscalar processor.

10

2. Discussion of the Related Art

Parallelism is extensively exploited in modern  
 computer designs. Among these designs are two distinct  
 architectures which are known respectively as the very  
 long instruction word (VLIW) architecture and the  
 superscalar architecture. A superscalar processor is a  
 computer which can dispatch one, two or more  
 instructions simultaneously. Such a processor  
 typically includes multiple functional units which can  
 independently execute the dispatched instructions. In  
 such a processor, a control logic circuit, which has  
 come to be known as the "grouping logic" circuit,  
 determines the instructions to dispatch (the  
 "instruction group"), according to certain resource  
 allocation and data dependency constraints. The task  
 of the computer designer is to provide a grouping logic  
 circuit which can dynamically evaluate such constraints  
 to dispatch instruction groups which optimally use the  
 available resources. A resource allocation constraint  
 can be, for instance, in a computer with a single  
 floating point multiplier unit, the constraint that no  
 more than one floating point multiply instruction is to  
 be dispatched for any given processor cycle. A  
 processor cycle is the basic timing unit for a  
 pipelined unit of the processor, typically the clock  
 period of the CPU clock. An example of a data

15

20

25

30

35

09583097-080299

dependency constraint is the avoidance of a "read-after-write" hazard. This constraint prevents dispatching an instruction which requires an operand from a register which is the destination of an write instruction dispatched earlier, but yet to be  
5     unretired.

662080" 2602560  
A VLIW processor, unlike a superscalar processor, does not dynamically allocate system resources at run time. Rather, resource allocation and data dependency  
10     analysis are performed during program compilation. A VLIW processor decodes the long instruction word to provide the control information for operating the various independent functional units. The task of the  
15     compiler is to optimize performance of a program by generating a sequence of such instructions which, when decoded, efficiently exploit the program's inherent parallelism in the computer's parallel hardware. The hardware is given little control of instruction sequencing and dispatch.

20     A VLIW computer, however, has a significant drawback in that its programs must be recompiled for each machine they run on. Such recompilation is required because the control information required by each machine is encoded in the instruction words. A  
25     superscalar computer, by contrast, is often designed to be able to run existing executable programs (i.e., "binaries"). In a superscalar computer, the instructions of an existing executable program are dispatched by the computer at run time according to the  
30     computer's particular resource availability and data integrity requirements. From a computer user's point of view, because existing binaries represent significant investments, the ability to acquire enhanced performance without the expense of purchasing  
35     new copies of binaries is a significant advantage.

In the prior art, to determine the instructions that go into an instruction group of a given processor

cycle, a superscalar computer performs the resource allocation and data dependency checking tasks in the immediately preceding processor cycle. Under this scheme, the computer designer must ensure that such resource allocation and data dependency checking tasks complete within their processor cycle. As the number of the functional units that can be independently run increases, the time required for performing such resource allocation and data dependency checking tasks grows more rapidly than linearly. Consequently, in a superscalar computer design, the ability to perform resource and data integrity analysis within a single processor cycle can become a factor that limits the performance gain of additional parallelism.

#### SUMMARY OF THE INVENTION

The present invention provides a central processing unit which includes a grouping logic circuit for determining simultaneously dispatchable instructions in an processor cycle. The central processing unit of the present invention includes such a grouping logic circuit and a number of functional units, each adapted to execute one or more specified instructions dispatched by the grouping logic circuit. The grouping logic circuit includes a number of pipeline stages, such that resource allocation and data dependency checks can be performed over a number of processor cycles. The present invention therefore allows dispatching a large number of instruction simultaneously, while avoiding the complexity of the grouping logic circuit from becoming limiting the duration of the central processing unit's processor cycle.

In one embodiment, the grouping logic circuit checks intra-group data dependency immediately upon receiving the instruction group. In that embodiment, all instruction in a group of instructions received in

a first processor cycle are dispatched prior to dispatching any instruction of a second group of instructions received at an processor cycle subsequent to said first processor cycle.

- 5        The present invention is better understood upon consideration of the detailed description below in conjunction with the accompanying drawings.

#### BRIEF DESCRIPTION OF THE DRAWINGS

- 10        Figure 1 is a block diagram of a CPU 100, in an exemplary 4-way superscalar processor of the present invention.

- 15        Figure 2 shows schematically a 4-stage pipelined grouping logic circuit 109 in the 4-way superscalar processor of Figure 1.

#### DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

- 20        An embodiment of the present invention is illustrated by the block diagram of Figure 1, which shows a central processing unit (CPU) 100 in an exemplary 4-way superscalar processor of the present invention. A 4-way superscalar processor fetches, dispatches, executes and retires up to four instructions per processor cycle. As shown in Figure 25 1, central processing unit 100 includes two arithmetic logic units 101 and 102, a load/store unit 103, which includes a 9-deep load buffer 104 and an 8-deep store buffer 105, a floating point adder 106, a floating point multiplier 107, and a floating point divider 108. 30 In this embodiment, a grouping logic circuit 109 dispatches up to four instructions per processor cycle. Completion unit 110 retires instructions upon completion. A register file (not shown), including numerous integer and float point registers, is provided 35 with sufficient number of ports to prevent contention among functional units for access to this register file during operand fetch or result write-back. In this

embodiment also, loads are non-blocking, i.e., CPU 100 continues to execute even though one or more dispatched load instructions have not complete. When the data of the load instructions are returned from the main memory, these data can be placed in a pipeline for storage in a second-level cache. In this embodiment, floating point adder 106 and floating point multiplier 107 each have a 4-stage pipeline. Similarly, load/store unit 103 has a 2-stage pipeline. Floating point divider 108, which is not pipelined, requires more than one processor cycle per instruction.

To simplify the discussion below, the state of CPU 100 relevant to grouping logic 109 is summarized by a state variable  $S(t)$ , which is defined below. Of course, the state of CPU 100 includes also other variables, such as those conventionally included in the processor status word. Those skilled in the art would appreciate the use and implementation of processor states. Thus, the state  $S(t)$  at time  $t$  of CPU 100 can be represented by:

Sub C2

$$S(t) = \{ALU_1(t), ALU_2(t), LS(t), LB(t), SB(t), FA(t), FM(t), FSD(t)\}$$

where  $ALU_1(t)$  and  $ALU_2(t)$  are the states, at time  $t$ , of arithmetic logic units 101 and 102 respectively;  $LS(t)$  and  $LB(t)$  are the states, at time  $t$ , of store buffer 105 and load buffer 104 respectively;  $FA(t)$ ,  $FM(t)$ , and  $FSD(t)$  are the states, at time  $t$ , of floating point adder 106, floating point multiplier 107 and floating point divider 108 respectively.

At any given time, the state of each functional unit can be represented by the source and destination registers specified in the instructions dispatched to the functional unit but not yet retired. Thus,

$$ALU_1 = \{ALU_1.rs1(t), ALU_1.rs2(t), ALU_1.rd(t)\}$$

where  $rs1(t)$ ,  $rs2(t)$  and  $rd(t)$  are respectively the first and second source registers, and the destination of registers of the instruction executing at time  $t$  in arithmetic logic unit 101.

Similarly, the state of arithmetic logic unit 102 can be defined as:

$$ALU_2 = \{ALU_2.rs1(t), ALU_2.rs2(t), ALU_2.rd(t)\}$$

10 Sub 3  
For pipelined functional units, such as floating point adder 106, the state is relatively more complex, consisting of the source and destination registers of the instructions in their respective pipeline. Thus, for the pipelined units, i.e., load/store unit 103, load buffer 104, store buffer 105, floating point adder 15 106, and floating point multiplier 107, their respective states, at time  $t$ ,  $LS(t)$ ,  $LB(t)$ ,  $SB(t)$ ,  $FA(t)$  and  $FM(t)$  can be represented by:

$$LS = \{LS.rs1_i(t), LS.rs2_i(t), LS.rd_i(t)\} \text{ for } i=\{1, 2\}$$

$$LB = \{LB.rs1_i(t), LB.rs2_i(t), LB.rd_i(t)\} \text{ for } i=\{1, 2, \dots, 9\}$$

20

$$SB = \{SB.rs1_i(t), SB.rs2_i(t), SB.rd_i(t)\} \text{ for } i=\{1, 2, \dots, 8\}$$

$$FA = \{FA.rs1_i(t), FA.rs2_i(t), FA.rd_i(t)\} \text{ for } i=\{1, \dots, 4\}$$

Sub 4  
Finally, floating point divider 108's state  $FSD(t)$

00583097-080299

$$FM = \{FM.rs1_i(t), FM.rs2_i(t), FM.rd_i(t)\} \text{ for } i=\{1, \dots, 4\}$$

can be represented by:

$$FDS = \{FDS.rs1_i(t), FDS.rs2_i(t), FDS.rd_i(t)\}$$

State variable  $S(t)$  can be represented by a memory element, such as a register or a content addressable memory unit, at either a centralized location or in a distributed fashion. For example, in the distributed approach, the portion of state  $S(t)$  associated with a given functional unit can be implemented with the control logic of the functional unit.

In the prior art, a grouping logic circuit would determine from the current state,  $S(t)$  at time  $t$ , the next state  $S(t+1)$ , which includes information necessary to dispatch the instructions of the next processor cycle at time  $t+1$ . For example, to avoid a read-after-write hazard, such a grouping circuit would exclude from the next state  $S(t+1)$  an instruction having an operand to be fetched from a register designated for storing a result of a yet incomplete instruction. As another example, such a grouping circuit would include in state  $S(t+1)$  no more than one floating point "add" instruction in each processor cycle, since only one floating point adder (i.e. floating point adder 106) is available. As discussed above, as complexity increases, the time required for propagating through the grouping logic circuit can become a critical path for the processor cycle. Thus, in accordance with the present invention, grouping logic circuit 109 is pipelined to derive, over  $\tau$  processor cycles, a future state  $S(t+\tau)$  based on the present state  $S(t)$ . The future state  $S(t+\tau)$  determines the instruction group to dispatch at time  $t+\tau$ . Pipelining grouping logic 109 is possible because, as demonstrated below, (i) the values of most state variables in the state  $S(t+\tau)$  can be

estimated from corresponding values of state  $S(t)$  with sufficient accuracy, and (ii) for those state variables for which values can not be accurately predicted, it is relatively straightforward to provide for all possible outcomes of state  $S(t+\tau)$ , or to use a conservative approach (i.e. not dispatching an instruction when such an instruction could have been dispatched) with a slight penalty on performance.

The process for predicting state  $S(t+\tau)$  is explained next. The following discussion will first show that most components of next state  $S(t+1)$  can be precisely determined from present state  $S(t)$ , and the remaining components of state  $S(t)$  can be reasonably determined, provided that certain non-deterministic conditions are appropriately handled. By induction, it can therefore be shown that future state  $S(t+\tau)$ , where  $\tau$  is greater than 1, can likewise be determined from state  $S(t)$ .

Since an instruction in floating point adder 106 or floating point multiplier 107 completes after four processor cycles and an instruction in load/store unit 103 completes after two processor cycles, the states FA, FM and LS at time  $t+1$  can be derived from the corresponding state  $S(t)$  at time  $t$ , the immediately preceding processor cycle. In particular, the relationship governing the source and destination registers of each instruction executing in floating point adder 106, floating point multiplier 107 and load/store unit 103 between time  $t+1$  and time  $t$  are:

$$rs1_i(t+1) = rs1_{i-1}(t), \quad \text{for } 1 \leq i \leq k$$

30

$$rs2_i(t+1) = rs2_{i-1}(t), \quad \text{for } 1 \leq i \leq k$$



$$rd_i(t+1) = rd_{i-1}(t), \quad \text{for } 1 < i < k$$

where  $k$  is the depth of the respective pipeline.

5 The state  $FSD(t+1)$  of floating point divider 108, in which the time required to execute an instruction can exceed an processor cycle, is determined from state  $FSD(t)$  by:

$$FSD(t+1) = FSD(t) \{ \text{if last stage} \} \text{ else null}$$

10 Whether or not floating point divider 108 is in its last stage can be determined from, for example, a hardware counter or a state register, which keep tracks of the number of processor cycles elapsed since the instruction in floating point divider 108 began execution.

15 In load buffer 104 and store buffer 105, since the pending read or write operation at the head of each queue need not complete within one processor cycle, the state  $LB(t+1)$  at time  $t+1$  cannot be determined from the immediately previous state  $LB(t)$  at time  $t$  with certainty. However, since state  $LB(t+1)$  can only  
20 either remain the same, or reflect the movement of the pipeline by one stage, two possible approaches to determine state  $LB(t+1)$  can be used. First, a conservative approach would predict  $LB(t+1)$  to be the same as  $LB(t)$ . Under this approach, when load buffer  
25 104 is full, an instruction is not dispatched until the pipeline in load buffer 106 advances. An incorrect prediction, i.e. a load instruction completes during the processor cycle of time  $t$ , this conservative approach leads to a penalty of one processor cycle,  
30 since a load instruction could have been dispatched at time  $t+1$ . Alternatively, a more aggressive approach provides for both outcomes, i.e. load buffer 104 advances one stage, and load buffer 104 remains the

5 same. Under this aggressive approach, grouping logic  
 109 is ready to dispatch a load instruction, such  
 dispatch to be enabled by a control signal which  
 indicates, at time  $t+1$ , whether a load instruction has  
 in fact completed. This aggressive approach requires  
 more a complex logic circuit than the conservative  
 approach.

10 Thus, the skilled person would appreciate that  
 state  $S(t+1)$  of CPU 100 can be predicted from state  
 $S(t)$ . Consequently, both the number of instructions  
 and the types of instructions that can be dispatched at  
 time  $t+1$  (i.e. the instruction group at time  $t+1$ ) based  
 on predicted state  $S(t+1)$  can be derived, at time  $t$ ,  
 from state  $S(t)$ , subject to additional handling based  
 15 on the actual state  $S_A(t+1)$  at time  $t+1$ .

The above analysis can be can be extended to allow  
 state  $S(t+r)$  at time  $t+r$  to be derived from state  $S(t)$   
 at time  $t$ . The instruction group at time  $t+r$  can be  
 derived from time  $t$ , provided that, for each  
 20 instruction group between time  $t$  and  $t+r$ , all  
 instruction from that instruction group must be  
 dispatched before any instruction from a subsequent  
 instruction group is allowed to be dispatched (i.e. no  
 instruction group merging).

25 Since instructions from different instruction  
 groups are not merged, intra-group dependencies and  
 inter-group dependencies can be checked in parallel.  
 The instructions are either fetched from an instruction  
 cache or an instruction buffer. An instruction buffer  
 30 is preferable in a system in which not all accesses  
 (e.g. branch instructions) to the instruction cache are  
 aligned, and multiple entry points in the basic blocks  
 of a program are allowed.

35 Once four candidate instructions for an  
 instruction group are identified, intra-group data  
 dependency checking can begin. Because of the  
 constraint against instruction group merging described

above, i.e., all instructions in an instruction group must be dispatched before an instruction from a subsequent instruction group can be dispatched, intra-group dependency checking can be accomplished in a pipelined fashion. That is, intra-group dependency checking can span more than one processor cycle and all inter-group dependency checking can occur independently of inter-group dependency checking. For the purpose of intra-group dependency check, each instruction group can be represented by:

$$\text{IntraS}(t) = \{rs1_i(t), rs2_i(t), rd_i(t), res_i(t)\} \\ \text{for } 0 \leq i < W-1$$

where  $W$  is the width of the machine, and  $res_i$  represents the resource utilization of instruction  $I$ . An example of a four-stage pipeline 200 is shown in Figure 2. In Figure 2, at first stage 201, as soon as the instruction group is constituted, intra-group dependency checking is performed immediately. Thereafter, at stage 202, resource allocation within the instruction group can be determined. At stage 203, intergroup decisions, e.g. resource allocation decisions taking into consideration resource allocation in previous instruction groups, are merged with the decisions at stages 201 and 202. For example, if the present instruction group includes an instruction designated for floating point divider 108, stage 203 would have determined at by this time if a previous instruction using floating point divider 108 would have completed by the time the present instruction group is due to be dispatched. Finally, at stage 204, non-deterministic conditions, e.g. the condition at store buffer 105, is considered. Dispatchable instructions are issued into CPU 100 at the end of stage 204.

The above detailed description is provided to

illustrate the specific embodiments of the present invention and is not intended to be limiting. Numerous variations and modifications within the scope of the present invention are possible. The present invention  
5 is defined by the following claims.

09583097.080299